Indexing Query Graphs to Speedup Graph Query Processing

Jing Wang
School of Computing Science
University of Glasgow, UK
j.wang.3@research.gla.ac.uk

Nikos Ntarmos
School of Computing Science
University of Glasgow, UK
nikos.ntarmos@glasgow.ac.uk

Peter Triantafillou
School of Computing Science
University of Glasgow, UK
peter.triantafillou@glasgow.ac.uk

ABSTRACT

Subgraph/supergraph queries although central to graph analytics, are costly as they entail the NP-Complete problem of subgraph isomorphism. We present a fresh solution, the novel principle of which is to acquire and utilize knowledge from the results of previously executed queries. Our approach, *i*GQ, encompasses two component subindexes to identify if a new query is a subgraph/supergraph of previously executed queries and stores related key information. *i*GQ comes with novel query processing and index space management algorithms, including graph replacement policies. The end result is a system that leads to significant reduction in the number of required subgraph isomorphism tests and speedups in query processing time. *i*GQ can be incorporated into any sub/supergraph query processing method and help improve performance. In fact, it is the only contribution that can speedup significantly both subgraph and supergraph query processing. We establish the principles of *i*GQ and formally prove its correctness. We have implemented *i*GQ and have incorporated it within three popular recent state of the art index-based graph query processing solutions. We evaluated its performance using real-world and synthetic graph datasets with different characteristics, and a number of query workloads, showcasing its benefits.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

General Terms

Design, Performance

Keywords

Graph query processing, indexing, query result caching

1. INTRODUCTION

Graph structured data are prevalent in many modern big data applications, ranging from chemical, bioinformatics, and other scientific datasets to social networking and social-based applications (such as recommendation systems). In biology, for example, there is a great need to model "structured interaction networks". These abound when studying species, proteins, drugs, genes, and molecular and chemical compounds, etc. In these graphs, nodes can model species, genes, etc. and edges reflect relationships between them. Molecular compounds, consisting of atoms and their bonds, are naturally modeled as graphs. Ditto for social networks, where nodes refer to people and edges to their relationships.

Developing systems and algorithms that can store, manage, and provide analysis over large numbers of (potentially large) graphs is a formidable challenge. Already, there exist several very large graph datasets. For instance, the PubChem[34] chemical compound dataset contains more than 35 million graphs and ChEBI[11] (Chemical Entities of Biological Interest) dataset contains more than half a million graphs. Further applications extend to software development and debugging[27] and to similarity searching in medical datasets[32]. As a result, a large number of graph data management systems, optimised for handling graph data, have emerged (e.g., Neo4J[4], InifiteGraph[20]). This is in addition to graph management systems designed by big data companies for their own purposes (e.g., Twitter's FlockDB[38], Google's Pregel[28]) and the list is continuously expanding. Hence, the demand for high performance data analytics in graph data systems is steadily increasing.

Central to graph analytics, is the need to locate patterns in dataset graphs. Informally, given a query graph, the system is called to identify which of the stored graphs in its dataset contain it (subgraph matching), or are contained in it (supergraph matching). This is a very costly operation as it entails the NP-Complete[14] problem of subgraph isomorphism and even its most popular solutions [9, 25, 39] are computationally very expensive. This problem is exacerbated when dealing with datasets storing large numbers of graphs, as the number of required subgraph isomorphism tests grows. Furthermore, performance deteriorates significantly with increasing graph sizes.

The key driver of our work is the realization that in many applications, it is natural to expect that queries submitted in the past share subgraph or supergraph relationships with queries of the future. As one example, consider chemical graph datasets, where queries use the graph representation of chemical entities. Such queries are naturally hierarchical: At the base, we see chemical elements. Then, there are graphs depicting chemical compounds (consisting of chemical elements), while there are also techniques to

create chemical compound clusters out of similar chemical compounds[34]. Similarly, in protein datasets there is also a hierarchy of queries for aminoacids, proteins, protein mixtures, proteins of uni-cell bacteria, all the way to those of multi-cell organisms. Finally, typical tools for social network analysis (SNA – e.g., Pajek[10]) provide the ability to produce graphs by filtering nodes and/or edges from other graphs. Using such graphs as queries in exploratory interactive SNA induces again the previous characteristic. For instance, consider the (query) graphs for analyzing friendship networks: such networks within the USA are subgraphs of friendship networks within North America, which in turn are subgraphs of the complete friendship network graph. The conclusion is that, in many applications, any query can itself be a subgraph or supergraph of a previously issued query. Up to now, this natural subgraph/supergraph relationship among queries has not been exploited.

2. PERSPECTIVES AND RELATED WORK

The problem of subgraph/supergraph query processing has been extensively studied. A prominent paradigm in the literature is the *filter then verify* paradigm. Essentially, this is an index-based class of methods. During indexing, the dataset graphs are reduced to their features (a *feature* being any substructure of a graph, be it path, tree, cycle, or arbitrary subgraph), which are inserted into an index structure (e.g., tree, trie, hash table, etc.). Given a query graph g, g is also decomposed into its features, following the same process as for dataset graphs. Then the index is searched for g's features; for subgraph queries, the set of graphs that contain all of said features are returned, whereas for supergraph queries the returned set consists of graphs all of whose features are contained in g's features. This set is called the *candidate set* and producing it constitutes the filtering stage of query processing.

All known algorithms guarantee that there will be no *false negatives*; that is, for subgraph (resp. supergraph) queries, all graphs in the dataset that can possibly contain (resp. are contained in) the query graph will be included in the candidate set. However, *false positives* are possible – not all graphs in the candidate set contain (resp. are contained in) the query graph. And herein lies the primary source of problems, since a subgraph subgraph isomorphism test must be performed against each graph in the candidate set, during the verification stage of query processing. The major focus of related work then is how to reduce the number of false positives, i.e., the number of unnecessary subgraph isomorphism tests.

Approaches in the literature can be classified along two dimensions: whether they employ (frequent) mining techniques or an exhaustive enumeration for the production of features, and based on the type of features of the dataset graphs they index (e.g., paths, trees, subgraphs). Note that exhaustive enumeration can yield huge indices and may take a prohibitively long time to do so. For this reason, all exhaustive enumeration approaches limit the size of features to a typically fairly small number of edges (i.e., 10 or less).

Mining-based approaches, both for supergraph queries ([5, 51, 46, 6, 52]) and subgraph queries (e.g., [41, 7, 52]) utilize techniques to mine for frequent (or *discriminating*, in [6]) (sub)graphs among the dataset graphs that are then indexed. Other mining-based approaches, like Tree+Δ[49] and TreePi[45] mine for and index frequent trees. Last, Lindex[43] and LWindex[44] utilize the frequent mining algorithms of previous approaches, and are thus able to index and query several feature types. Typically such approaches tend to mine for more complex structures, which presents a trade-off between the complexity and time required for the indexing process vis-a-vis the potential for higher pruning power during query processing. However, numerous related performance studies [21, 12, 15, 17, 22] have shown that feature-mining approaches tend to be comparatively worse performers.

On the other hand, SING[12], GraphGrep[16] and GraphGrepSX[3] perform exhaustive enumeration, listing all paths of dataset graphs up to a certain path length. Similarly, CT-Index[22] indexes trees and cycles, whereas Grapes[15] indexes paths along with location information.

A different approach, which does not index features as above, is presented in gCode[53]. For each graph G in the graph dataset, gCode computes a signature per vertex of G (essentially reflecting the vertex's neighbourhood) and then computes a signature for G itself. The latter is a tree structure combining the signatures of all its vertices.

With respect to the verification stage, approaches also differ on how this is performed. In some works, verification is performed by applying any (exact) subgraph isomorphism algorithm of choice (see [25] for a detailed insightful comparative evaluation) after the filtering stage. Indeed, this can be the default choice for all approaches and there is a large variety of subgraph isomorphism algorithms available. Most such algorithms are influenced by Ullman's early work [39]. Arguably, the algorithm that is now the most widely used is the VF2[9] algorithm. Last, several approaches store and utilize location information in their index to achieve further filtering ([45, 12, 15]).

Recent performance studies [17, 21] have shown that CT-Index[22] and Grapes[15] are high performing approaches. CT-Index[22] is based on deriving canonical forms for the (tree, cycle) features of a graph G, to the fact that for trees and cycles finding string-based canonical forms can be done in linear time (unlike general graphs). These string representations of a graph's features are then hashed into a bitmap structure per graph G. Checking whether a query graph g can possibly be a subgraph of a graph G, can be done with simple bitwise operators between the bitmap of g and that of G (as supergraphs must contain all features of a subgraph). Last, its verification stage is then based on VF2.

Grapes[15] is designed to exploit parallelism available in multi-core machines. It exhaustively enumerates all paths (up to a maximum length), which are then inserted into a trie with their location information. This operation is performed in parallel by several threads, each of which works on a portion of the graph, producing its own trie, and subsequently all tries are merged together to form the path index of a graph. Grapes then computes (typically) small connected components of graphs in the candidate set, on which the verification (subgraph isomorphism test) is performed.

An insightful discussion and comparative performance evaluation of several indexing techniques for subgraph query processing (published prior to 2010) can be found in [17]. Furthermore, in [21] we presented a systematic performance and scalability study of several older as well as current state-of-the-art index-based approaches for subgraph query processing. We are not aware of similar in-depth studies of solutions to supergraph query processing; however, [44] pro-

vides a concise overview of related approaches.

On a related note, recent work also deals with graph querying against historical graphs, identifying subgraphs enduring graph mutations over time [35], which can be viewed as a variation whereby graph snapshots in time can be viewed as different graphs. Also, the research community has recently started looking into subgraph queries against a single, very large graph consisting of possibly billions of nodes[36]. To accelerate the query processing, SPath[48] proposes a path-at-a-time fashion, which proves to be more efficient than traditional vertex-at-a-time methods, whereas [36] makes use of a memory cloud and [33, 1, 24] exploit MapReduce. In this subgraph querying problem for the single large graph setting, the goal is to expedite the subgraph isomorphism itself, whereas in the setting with many dataset graphs, the target of subgraph querying problem is to minimize the number of isomorphism tests that need to be performed. Our work focuses on the latter setting and leaves for future work the application of our ideas to the former setting.

There has also been considerable work on approximate graph pattern matching. Relevant techniques (e.g., [22, 18, 37, 40, 42, 47, 50, 33, 13]) perform subgraph matching with support for wildcards and/or approximate matches. These solutions are not directly related to our work, as we expedite exact index-based subgraph/supergraph query processing.

Caching of the results of path/tree queries has been explored in XML databases[26, 2, 29]. The problem we focus on is considerably different, as the queries we deal with are in the form of graphs (not just paths/trees), thus entailing the NP-Complete problem of subgraph isomorphism. Furthermore, in our setting queries retrieve stored graphs that contain the query graph (subgraph queries) or are contained in it (supergraph queries), and we exploit both supergraph and subgraph relationships among queries themselves, as opposed to only subsumption (i.e., supergraph) relationships. Moreover, our graph replacement policy also takes into account the subgraph isomorphism costs, as opposed to just the size or popularity of cached queries.

Last, [23] presents a cache for targeted historical queries against a large social graph. In this case, each query is centered around a uniquely identified node in the social graph, and the objective is to avoid maintaining and/or reconstructing complete snapshots of the social graph, but to instead use a set of static "views" – i.e., snapshots of neighborhoods of nodes – to rewrite incoming queries. [23] does not deal with subgraph/supergraph query processing; rather, the nature of the queries means that containment can be decided by simply measuring the distance of the central query node to the center of each view, while also taking into account the diameter of these two graphs. Furthermore, the authors do not provide a cache replacement strategy, but rather an algorithm to compute the optimal cache contents given a set of queries. *i*GQ could well be used to both generalise and expedite query processing in [23].

2.1 The iGQ Perspective

In this work we offer a new perspective and a strategy for improving subgraph/supergraph query processing performance and scalability. Our approach rests on the following three observations: First, in related works there exists an implicit assumption that graph queries will be similarly structured to the dataset graphs. In general this is not guaranteed to hold (e.g., in exploratory analytics), and when query graphs have no match in the dataset graphs, query processing cannot benefit at all from indexes that are solely constructed on dataset graphs. Second, even when query graphs have matches against dataset graphs, the system performs expensive computations during query processing and simply throws away all (painstakingly and laboriously) derived knowledge (i.e.,previous querying result). Third, the success of known approaches depends on and exploits the fact that dataset graphs share features (e.g., when mining for frequent features) and/or that dataset graph features contain or are contained in other graph features (e.g., when using tries to index dataset graph features). However, they completely fail to investigate and exploit such similarities between query graphs.

As mentioned, it is natural in many applications for new queries to bear subgraph/supergraph relationships with previously issued queries. Our efforts in this work centre on exploiting this characteristic to further improve the performance of query processing. Therefore, instead of "mining" only the stored graphs and creating relevant indexes on them, we also "mine" *query graphs* and accumulate the knowledge produced by the system when running queries, creating a *query index* in addition to the *dataset index*. Our insights identify which is the relevant accumulated knowledge and how to exploit it during query processing in order to further reduce the number of subgraph isomorphism tests. *i*GQ can accommodate any proposed index for sub or supergraph query processing and help expedite both query types.

2.2 Contributions

The contributions of this work are that we:

- Provide a new perspective to the problem of subgraph/ supergraph query processing, with insights as to how the work performed by the system when executing queries can be appropriately managed to improve the performance of future queries.
- Detail the *i*GQ approach, based on a query index structure and associated query processing algorithms, which can reduce the number of isomorphism tests performed during query processing.
- Present the *i*GQ framework, showing how to incorporate *i*GQ within existing approaches, and the two *i*GQ components: a subgraph query index and a supergraph query index. The subgraph index of *i*GQ can be based on any existing subgraph index (over query graphs, not dataset graphs). The supergraph index on the other hand is a new index to swiftly determine supergraph status between new and previous queries.
- Address the issue of index space management, providing mechanisms for index updates and a graph replacement policy, deciding contents of query index.
- Implement *i*GQ, incorporate it within three popular approaches for graph query processing, and provide experimental results using real-world datasets and a number of query workloads, showcasing *i*GQ's benefits against competitive state of the art methods.

3. PROBLEM FORMULATION

We consider undirected labeled graphs. For simplicity, we assume that only vertices have labels; all our results straightforwardly generalize to graphs with edge labels.

DEFINITION 1. *A labeled graph* $G = (V, E, l)$ *consists of*

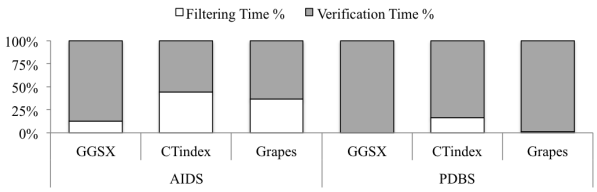

Figure 1: Dominance of the Verification Time on the Overall Query Processing Time of Three Subgraph Querying Algorithms on Two Different Real-world Graph Datasets

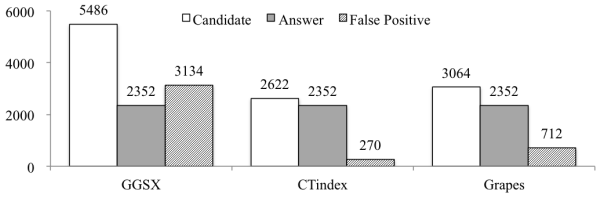

Figure 2: Average Number of Candidates, Answer Set Size, and False Positives in the AIDS Dataset

a set of vertices $V(G)$ and edges $E(G) = \{(u, v), u \in V, v \in V\}$, and a function $l : V \rightarrow U$, where U is the label set, defining the domain of labels of vertices.

A sequence of vertices $(v_0, \ldots, v_n)$ *s.t.* $\exists(v_i, v_{i+1}) \in E$, constitutes a path of length n. A simple path is a path where no vertices are repeated. A cycle is a path of length $n > 1$, where $v_0 = v_n$. A simple cycle is a cycle with no repeated vertices (other than v_0 and v_n). A connected graph is one where there exists a path between any pair of its vertices.

DEFINITION 2. *A graph $G_i = (V_i, E_i, l_i)$ is subgraph-isomorphic to a graph $G_j = (V_j, E_j, l_j)$, (by abuse of notation) denoted by $G_i \subseteq G_j$, when there exists an injection $\phi : V_i \rightarrow V_j$, such that $\forall(u, v) \in E_i, u, v \in V_i, \Rightarrow (\phi(u), \phi(v)) \in E_j$ and $\forall u \in V_i, l_i(u) = l_j(\phi(u))$.*

Informally, there is a subgraph isomorphism $G_i \subseteq G_j$ if G_j contains a subgraph that is isomorphic to G_i. In this case, we say that G_i is a subgraph of (or contained in) G_j, or inversely that G_j is a supergraph of (contains) G_i (denoted by $G_j \supseteq G_i$).

DEFINITION 3. *The subgraph querying problem entails a set $D = \{G_1, \ldots, G_n\}$ containing n graphs, and a query graph g, and determines all graphs $G_i \in D$ such that $g \subseteq G_i$.*

DEFINITION 4. *The supergraph querying problem entails a set $D = \{G_1, \ldots, G_n\}$ containing n graphs, and a query graph g, and determines all graphs $G_i \in D$ such that $g \supseteq G_i$.*

The iGQ index, $\mathbb{I}$, will be called to index the features of query graphs; then we shall say that query graph g is indexed by iGQ and (by abuse of notation) denote it by $g \in \mathbb{I}$. We denote with $\mathbb{I}_{sub}(g)$ all query graphs currently contained in $\mathbb{I}$ that are supergraphs of g (answers to g, if g was a subgraph query); i.e., $\mathbb{I}_{sub}(g) = \{G \mid G \in \mathbb{I} \wedge g \subseteq G\}$. Similarly, we denote with $\mathbb{I}_{super}(g)$ all query graphs currently contained in $\mathbb{I}$ that are subgraphs of g (answers to g, if g was a supergraph query); i.e., $\mathbb{I}_{super}(g) = \{G \mid G \in \mathbb{I} \wedge g \supseteq G\}$.

4. iGQ PRINCIPLES

We firstly discuss our findings from experiments we ran regarding the major performance obstacles we need to overcome if we are to bring about further query processing time reductions. Subsequently, we present the iGQ framework, followed by an explanation of how the components of iGQ are utilized for further performance improvements, and related formal proofs of correctness. As mentioned, so far related work has not considered benefiting from the execution of previous queries. Thus, despite devoting a lot of resources to such queries, the results derived cannot be put to good use to improve performance of future subgraph queries.

4.1 Insights

We report on the fundamentals of the performance of three state of the art approaches, GraphGrepSX[3] (GGSX), Grapes[15], and CT-Index[22], over three real datasets and one synthetic dataset with different characteristics. These characteristics will be presented in detail in the experimental evaluation section. Briefly, AIDS represents a graph DB consisting of 40,000 very small, sparse graphs, while PDBS is a graph dataset containing 600 large graphs. Please note that the way the queries were generated is standard among related work [15, 22].

Subgraph Query Performance: Where Does Time Go?

There are two key components of the overall query processing time: filtering time (to process the index and produce the candidate set) and verification time (to perform the verification of all candidate graphs). Fig. 1 shows what percentage of the total query processing time is attributed to each component.

The dominance of the verification step is clear. This holds across the three different approaches that employ different indexing methods and utilize different strategies for cutting down the cost of subgraph isomorphism. Recall that subgraph isomorphism performance is highly sensitive to the size of both the input graph and the stored graph. Hence, we would expect that for smaller stored graphs (as in the AIDS dataset) the verification step would be much faster. Notably, however, even when graphs are very small, the verification step is the biggest performance inhibitor and as graphs become larger (e.g., PDBS) the verification step becomes increasingly responsible for nearly the total query processing time. Of course, given the NP-Completeness of subgraph isomorphism, one would expect that verification would dominate, especially for large graphs. But the fact that even with very small graphs this holds is noteworthy.

Filtering Power: Is It Good Enough?

The second fundamental point pertains to how one can reduce the verification cost. Related works highlight that their approaches prove to be very powerful in terms of filtering out the vast majority of DB graphs. In Figures 2 and 3 we show our results with respect to the average size of candidate sets and of the answer set, as well as the average number of false positives for the AIDS and PDBS datasets.

First, note that different algorithms behave differently in different datasets (e.g., Grapes significantly outperforms CT-Index in PDBS while the reverse holds for AIDS). Second, note that despite the powerful filtering of an approach, when the DB contains a large number of graphs (see Figure 2) in absolute numbers, there is a very large number of unnecessary subgraph isomorphism tests (i.e., false positives)

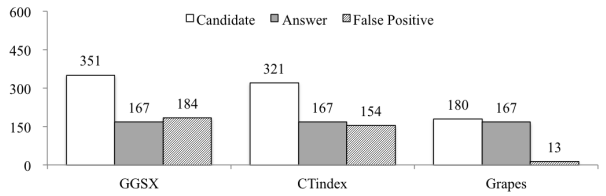

Figure 3: Average Number of Candidates, Answer Set Size, and False Positives in the PDBS Dataset

that is required. The above two combined imply that even the best algorithm will suffer from a large number of unnecessary subgraph isomorphism tests under some datasets.

Turning our attention to Figure 3 we see that for DBs with medium to small number of graphs, the high filtering power can indeed result in requiring only a relatively small number of subgraph isomorphism tests. However, considerable percentages of false positives can appear in the candidate sets of even top-performing algorithms; e.g., CT-Index, which exhibited the best filtering in the AIDS dataset, has an almost 50% false positive ratio in the PDBS dataset. Furthermore, not all subgraph isomorphism tests for the graphs in the candidate set are equally costly. As the cost of subgraph isomorphism testing depends on the size of the graph, the larger graphs in the candidate set contribute a much greater proportion of the total cost of the verification step. Note that, naturally, false positive graphs tend to be the largest graphs in the DB, since these have a higher probability to contain all features of query graphs.

Note that we placed emphasis on the number of unnecessary subgraph isomorphism tests (i.e., the false positives), as we can improve filtering further by reducing this number. However, this is not the only source of possible improvements. As we shall show later, *i*GQ can improve on the number of subgraph isomorphism tests even beyond this, by exploiting knowledge gathered during query execution.

The insights that can be drawn are as follows:

- Despite the fact that state of the art techniques (based on indexing features of DB graphs) can enjoy high filtering capacity, there is still large room for improvement, as even the best approaches may perform large numbers of unnecessary subgraph isomorphism tests.
- Improving further the filtering power of approaches can significantly improve query processing time, as this will reduce the number of subgraph isomorphism tests, which dominates the overall querying time.
- Even approaches that are purported to enjoy great filtering powers, can behave much more poorly under different datasets.
- Unnecessary subgraph isomorphism tests are not solely caused by false positives; even graphs in the candidate set that are true positives can be unnecessarily tested if the system fails to exploit this knowledge (accrued by previous query executions).

4.2 The iGQ Framework

*i*GQ aims to augment the functionality and benefits offered by any one of the subgraph and/or supergraph indexing methods in the literature. Let us call the chosen method $\mathbb{M}$. The *i*GQ framework consists of method $\mathbb{M}$ and the two components of $\mathbb{I}$, $\mathbb{I}_{sub}$ and $\mathbb{I}_{super}$. For the sake of simplicity, we shall first describe the operation of *i*GQ when $\mathbb{M}$ is a method for subgraph query processing (denoted $\mathbb{M}_{sub}$). Initially, method $\mathbb{M}_{sub}$ builds its graph dataset index as per usual. The *i*GQ index, $\mathbb{I}$, starts off empty; it is then populated as queries arrive and are executed by $\mathbb{M}_{sub}$.

Upon the arrival of a query g, the query processing process is parallelized. One thread uses method $\mathbb{M}_{sub}$'s algorithms and indexing structure to breakdown the query graph into its features, and uses its index to produce a candidate set of graphs, $CS(g)$, as usual. Additionally, $\mathbb{I}$ will obtain as many of the intermediate and final results from method $\mathbb{M}$'s execution as possible; e.g., it will obtain the features of the query graph, to be compared to those stored in $\mathbb{I}$ (from previously-executed queries). At this point, two separate threads will be created: one will check whether the query graph is a subgraph of previous query graphs and the other will check whether it is a supergraph of previous query graphs. These cases yield different opportunities for optimization and are discussed separately below.

In the following we proceed to describe the function of each component of the *i*GQ framework and how it is all brought together. For the formal proofs of correctness that follow, for simplicity, we make the following assumptions.

Assumptions. The *i*GQ index components, $\mathbb{I}_{sub}$ and $\mathbb{I}_{super}$ work correctly. That is:

$$G \in \mathbb{I}_{sub}(g) \Rightarrow g \subseteq G \quad (1)$$

and

$$G \in \mathbb{I}_{super}(g) \Rightarrow g \supseteq G \quad (2)$$

We will prove that these assumptions hold in sections 6.1 and 6.2.

4.2.1 The Subgraph Case: $\mathbb{I}_{sub}$

This case occurs when a new query g is a subgraph of a previous query G. When G was executed by the system, the $\mathbb{I}_{sub}$ component of *i*GQ indexed G's features. Additionally, *i*GQ stored the results computed by $\mathbb{M}_{sub}$ for G.

Fig. 4 depicts an example for the subgraph case of *i*GQ. A new query g is "sent" to method $\mathbb{M}_{sub}$'s graph index, producing a candidate set, $CS(g)$, which in this case contains the four graphs $\{g_1, g_2, g_3, g_4\}$. Similarly, g is "sent" to the *i*GQ subgraph component, $\mathbb{I}_{sub}$, from where it is determined that there exists a previous query G, such that $g \subseteq G$. *i*GQ then retrieves the answer set, $Answer(G)$ (previously produced by method $\mathbb{M}_{sub}$ and indexed by $\mathbb{I}_{sub}$); in this case, $Answer(G) = \{g_1, g_2\}$. The reasoning then proceeds as follows. Consider graph $g_1 \in CS(g)$. Since from $\mathbb{I}_{sub}$ it has been concluded that $g \subseteq G$ and from the answer set of G we know that $G \subseteq g_1$, it necessarily follows that $g \subseteq g_1$. Similarly, we conclude that $g \subseteq g_2$. Hence, there is no point in testing g for subgraph isomorphism against g_1 or g_2, as the answer is already known. Therefore, one can safely subtract graphs g_1, g_2 from $\mathbb{M}_{sub}$'s candidate set, and test only the remaining graphs (reducing the number of subgraph isomorphism tests in this example by 50%). After the verification stage, g_1, g_2 are added to the final answer set.

In the general case, g may be a subgraph of multiple previous query graphs G_i in $\mathbb{I}_{sub}$. Following the above reasoning, we can safely remove from $CS(g)$ all graphs appearing in the answer sets of all query graphs G_i, as they are bound to be supergraphs of g; that is, the set of graphs submitted by

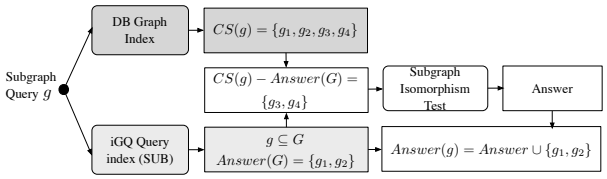

Figure 4: iGQ Processing of a Subgraph Query in the Subgraph Case

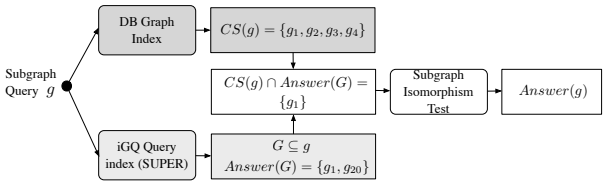

Figure 5: iGQ Processing of a Subgraph Query in the Supergraph Case

iGQ for subgraph isomorphism testing is given by:

$$CS_{sub}(g) = CS(g) \setminus \bigcup_{G_i \in \mathbb{I}_{sub}(g)} Answer(G_i) \quad (3)$$

Finally, if $Answer_{sub}(g)$ is the subset of graphs in $CS_{sub}(g)$ verified to be containing g through subgraph isomorphism testing, the final answer set for query g will be:

$$Answer(g) = Answer_{sub}(g) \cup \bigcup_{G_i \in \mathbb{I}_{sub}(g)} Answer(G_i) \quad (4)$$

LEMMA 1. *The iGQ answer in the subgraph case does not contain false positives.*

PROOF. Assume that a false positive was produced by iGQ; particularly, consider the first ever false positive produced by $\mathbb{I}_{sub}$, i.e., for some query g, $\exists G_{FP}$ such that $g \nsubseteq G_{FP}$ and $G_{FP} \in Answer(g)$. Note that G_{FP} cannot be in $Answer_{sub}(g)$, as the latter contains only those graphs from $CS_{sub}(g)$ that have been verified to be supergraphs of g after passing the subgraph subgraph isomorphism test, and hence $g \nsubseteq G_{FP} \Rightarrow G_{FP} \notin Answer_{sub}(g)$. Therefore, by formula (4), $G_{FP} \in Answer(g) \Rightarrow \exists G$ such that $G \in \mathbb{I}_{sub}(g)$ and $G_{FP} \in Answer(G)$. But (by formula (1)) $G \in \mathbb{I}_{sub}(g)$ $\Rightarrow g \subseteq G$, and $G_{FP} \in Answer(G) \Rightarrow G \subseteq G_{FP}$. Thus $g \subseteq G_{FP}$ (a contradiction). □

LEMMA 2. *iGQ in the subgraph case does not introduce false negatives.*

PROOF. Assume that a false negative was produced by iGQ; particularly, consider the first ever false negative produced by $\mathbb{I}_{sub}$, i.e., for some query g, $\exists G_{FN}$ such that $g \subseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. As method $\mathbb{M}_{sub}$ is assumed to be correct, it cannot produce any false negatives when processing query g, hence $g \subseteq G_{FN} \Rightarrow G_{FN} \in CS(g)$. Then, the only possibility for error is that G_{FN} was removed using formula (3); i.e., $G_{FN} \notin CS_{sub}(g)$. That implies that $\exists G$ such that $G \in \mathbb{I}_{sub}(g)$ and $G_{FN} \in Answer(G)$. But then, by formula (4), G_{FN} will be added to $Answer_{sub}(g)$ and thus $G_{FN} \in Answer(g)$ (a contradiction). □

THEOREM 1. *The iGQ answer in the subgraph case of query processing is correct.*

PROOF. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 1 and 2. □

4.2.2 The Supergraph Case: $\mathbb{I}_{super}$

This case occurs when a new query g is a supergraph of a previous query G. Fig. 5 depicts an example for the supergraph case of iGQ. Again, the subgraph query processing method $\mathbb{M}_{sub}$ produces a candidate set, $CS(g)$ that, say, contains four graphs $\{g_1, g_2, g_3, g_4\}$. Running g through $\mathbb{I}_{super}$, it is determined that there exists a previous query graph G such that $G \subseteq g$. Also $\mathbb{I}_{super}$ supplies the stored answer set for G, $Answer(G) = \{g_1, g_{20}\}$.

The reasoning then proceeds as follows. Consider graph $g_2 \in CS(g)$. We know from $\mathbb{I}_{super}$ that $g_2 \notin Answer(G)$. Now, if $g \subseteq g_2$ were to indeed be true, since $G \subseteq g$, then it must also hold that $G \subseteq g_2$; that is, $Answer(G)$ would have to contain g_2 as well, which is a contradiction. Therefore, it is safe to conclude that $g \nsubseteq g_2$ and thus g_2 can be safely removed from $CS(g)$. Similarly, we can also safely remove graphs g_3, g_4 from $CS(g)$, reducing in this case the number of required subgraph isomorphism tests by 75%. Thus, only g_1 needs to be isomorphism-tested in this example.

In the general case, g may be a supergraph of multiple previous query graphs G_i in $\mathbb{I}_{super}$. By the above reasoning, only those graphs appearing in the answer sets of all queries G_i may actually be supergraphs of g; thus the set of graphs submitted by iGQ for subgraph isomorphism testing is:

$$CS_{super}(g) = CS(g) \cap \bigcap_{G_i \in \mathbb{I}_{super}(g)} Answer(G_i) \quad (5)$$

The final answer produced for query g by iGQ, $Answer(g)$, will be the subset of graphs in $CS_{super}(g)$ that have been verified by the subgraph isomorphism test.

LEMMA 3. *The iGQ answer in the supergraph case does not contain false positives.*

PROOF. This trivially follows by construction as all graphs in $Answer(g)$ have passed through subgraph isomorphism testing at the final stage of processing. □

LEMMA 4. *The iGQ answer in the supergraph case does not introduce false negatives.*

PROOF. Assume false negatives are possible and consider the first ever false negative produced by $\mathbb{I}_{super}$; i.e., for some query g, $\exists G_{FN}$ such that $g \subseteq G_{FN}$ and $G_{FN} \notin Answer(g)$. Method $\mathbb{M}_{sub}$ does not produce in its candidate set any false negatives (as will be formally proven shortly), hence $G_{FN} \in CS(g)$. Then, the only possibility for error is for iGQ to have removed graph G_{FN} from $CS_{super}(g)$ with formula (5). This implies that $\exists G$ such that $G \in \mathbb{I}_{super}(g)$ and $G_{FN} \notin Answer(G)$. But since $G \in \mathbb{I}_{super}(g)$, by equation (2), $G \subseteq g$, and then $g \subseteq G_{FN} \Rightarrow G \subseteq G_{FN} \Rightarrow G_{FN} \in Answer(G)$ (a contradiction). □

THEOREM 2. *The iGQ answer in the supergraph case of query processing is correct.*

PROOF. There are only two possibilities for error; iGQ can produce false negatives or false positives. The theorem then follows straightforwardly from Lemmas 3 and 4. □

4.3 iGQ and Optimal Performance

There are two special cases that warrant further emphasis, since they introduce the greatest possible benefits.

First, note that *i*GQ can easily recognize the case where a new query, g, is exactly the same as a previous query contained in $\mathbb{I}$. Specifically, this holds when $\exists G \in \mathbb{I}$ such that $g \subseteq G$ or $g \supseteq G$, and g and G have the same number of nodes and edges. When this holds, since $\mathbb{I}$ stores the result for G, we can return directly and completely avoid the subgraph isomorphism testing as the actual result for g is known! As the subgraph isomorphism test dominates the query execution time, this is expected to be a large performance improvement.

Second, consider the supergraph part of *i*GQ. If $\exists G \in \mathbb{I}_{super}(g)$ such that $G \subseteq g$ and $Answer(G) = \emptyset$, then we can completely omit the verification stage again: If there were a dataset graph G' such that $g \subseteq G'$, since $G \subseteq g$ we would conclude that $G \subseteq G'$, which necessarily implies that $G' \in Answer(G)$, which contradicts the fact that $Answer(G) = \emptyset$. Thus, no such graph G' can exist and it is safe to stop query processing at this stage.

4.4 iGQ and Supergraph Query Processing

As mentioned earlier, *i*GQ can expedite both subgraph *and* supergraph query processing. In the latter case, the components of *i*GQ ($\mathbb{I}_{sub}$, $\mathbb{I}_{super}$) remain unchanged, but the handling of the return answer sets is the exact inverse of what happens for subgraph queries. Briefly, given a supergraph query processing method $\mathbb{M}_{super}$ and a supergraph query g, the union of the answer sets of graphs in $\mathbb{I}_{super}(g)$ are removed from $CS_{super}(g)$ and added to $Answer_{super}(g)$ to produce the final answer, and the graphs not appearing in the intersection of the answer sets of graphs in $\mathbb{I}_{sub}(g)$ are completely subtracted from $CS_{sub}(g)$. Also, the first optimal case mentioned above still holds, but the second optimal case is inversed with the processing terminating when $\exists G \in \mathbb{I}_{sub}(g)$ such that $Answer(G) = \emptyset$. The intuition behind this design and the proof of correctness of *i*GQ for supergraph query processing, follow the same reasoning as above and are omitted for space reasons. The elegance afforded by the double use of *i*GQ is unique.

5. iGQ INDEX SPACE MANAGEMENT

As queries arrive continuously and the space to store $\mathbb{I}$ is finite, *i*GQ requires methods for (i) efficiently handling this space and (ii) ensuring that it is best utilized, keeping those query graphs that increase its performance impact.

5.1 iGQ Graph Replacement Policy

Our replacement policy differs fundamentally from standard replacement policies: Unlike traditional cache replacement, whereby replacing a page or a file block saves one IO, different graphs in $\mathbb{I}$ bring about different benefits, as is shown below. We identify three key principles.

Increase the use of iGQ index. $\mathbb{I}$ should contain popular graphs; this is typical of all replacement algorithms. We define the popularity of a graph g as $P(g) = \frac{H(g)}{M(g)}$, where $H(g)$ is the number of times a graph $g \in \mathbb{I}$ has been found to be a subgraph or supergraph of query graphs (*hit*), and $M(g)$ is the total number of all queries processed since g was added to the *i*GQ index. In essence, this models the fraction of queries affected over time by g being in $\mathbb{I}$.

Reduce the number of subgraph isomorphism tests. Ideal graphs for $\mathbb{I}$ are graphs that bring about the greatest possible reductions in the number of executed subgraph isomorphism tests. Let $R(g)$ be the total number of graphs removed from the candidate sets of incoming queries because of g being in $\mathbb{I}$. Then this component is computed as $\frac{R(g)}{H(g)}$ – the per-*hit* average number of subgraph isomorphism tests alleviated by g.

Reduce the cost of each subgraph isomorphism test. A graph $g \in \mathbb{I}$ is more desirable if it helps avoid subgraph isomorphism tests on the biggest graphs from $\mathbb{M}$'s CS. This is so since we also wish to remove from CS graphs with expensive subgraph isomorphism tests. We denote by $C(g)$ the total cost of the subgraph subgraph isomorphism tests alleviated as a result of g being in $\mathbb{I}$. In order to estimate this value, we extend the asymptotic complexity analysis of [8] to the case of subgraph isomorphism. Specifically, given graphs with L labels, graph g' with n nodes, and graph G_i with $N_i \geq n$ nodes, the cost $c(g', G_i)$ of subgraph isomorphism of g' against G_i is given by:

$$c(g', G_i) = \frac{N_i \times N_i!}{L^{n+1} \times (N_i - n)!}$$

$C(g)$ is then computed as the sum over all $c(g', G_i)$, for all g' whose $CS(g')$ was reduced by removing G_i as a result of g being in $\mathbb{I}$, and $\frac{C(g)}{R(g)}$ gives the average cost reduction per alleviated test.

Ideal graphs for $\mathbb{I}$ are those that could help future queries as much as possible. To quantify such a contribution, we introduce the notion of graph *utility*, $U(g)$, defined as:

$$U(g) = \frac{H(g)}{M(g)} \times \frac{R(g)}{H(g)} \times \frac{C(g)}{R(g)} = \frac{C(g)}{M(g)}$$

That is, the utility of a graph g in *i*GQ is equal to the probability of g being used for an incoming query (i.e., being *hit*), times the average savings in number of subgraph isomorphism tests per such hit, times the average cost for a single subgraph isomorphism test. The replacement policy is then based on this, with the graph with the smallest $U(g)$ being evicted.

5.2 iGQ Index Maintenance Policy

For all graphs in $\mathbb{I}$ we maintain the metadata mentioned above (i.e., $C(g), M(g)$). Additionally, we store the actual query graphs that are indexed by $\mathbb{I}$ in a separate store coined $\mathbb{I}_{graphs}$. To facilitate index updates without interfering with query processing performance, we employ the concepts of *query window size*, W, and *cache size*, $\mathcal{C}$, with $W \leq \mathcal{C}$. As new graph queries arrive, they are processed as outlined above, update the metadata for graphs in $\mathbb{I}$, and are inserted into a temporary storage $\mathbb{I}_{temp}$. When W new queries have been processed, we consult the metadata to locate the W graphs in $\mathbb{I}$ with the lowest utility values. The graph data for those graphs is removed from $\mathbb{I}_{graphs}$ and replaced by the graphs in $\mathbb{I}_{temp}$. The latter is then emptied, and a "shadow" index, $\mathbb{I}_{shadow}$, is built over graphs in $\mathbb{I}_{graphs}$. Incoming queries keep being served by $\mathbb{I}$ and updating its metadata. When the shadow indexing is over, $\mathbb{I}_{shadow}$ replaces $\mathbb{I}$ (with a pointer swap). Finally, metadata for graphs removed from $\mathbb{I}$ is also removed from the metadata store ($C(g), M(g)$).

6. iGQ ALGORITHMS AND STRUCTURES

The proofs of correctness provided by the previous section, assume that $\mathbb{I}_{sub}$ and $\mathbb{I}_{super}$ provide correct results (recall formulas (1) and (2)). We shall now discuss the associated mechanisms and prove that they hold.

Algorithm 1 The Supergraph Index in iGQ

```
1: Input: Set ℚ of (previous) queries g_1, g_2, ..., g_n
2: Output: Supergraph index of previous queries 𝕀_super
3:
4: Initialize 𝕀_super to an empty TRIE
5: for all g_i ∈ ℚ do
6:     Extract all features of g_i and insert them in set F(g_i)
7:     NF[g_i] = |F(g_i)|
8:     for all features f ∈ F(g_i) do
9:         o = number of occurrences of f in g_i
10:        𝕀_super.insert(f, {g_i, o})
11:    end for
12: end for
13: return 𝕀_super
```

6.1 Finding Supergraphs in $\mathbb{I}_{sub}$

This case represents a microcosm of our original problem, where instead of indexing and querying dataset graphs, we index and query previous query graphs. Hence, any approach from the related works can be adapted for this purpose. Actually, as *i*GQ can complement any existing approach, $\mathbb{M}_{sub}$, we can utilize $\mathbb{M}_{sub}$'s method for subgraph query processing for the subgraph case of *i*GQ, or any other method appropriate for *i*GQ's special characteristics (i.e., relatively small set of small graphs). Note that the assumed correct method $\mathbb{M}_{sub}$ precludes false negatives and subgraph isomorphism testing of all candidates precludes false positives. Hence, formula (1)'s assumption is trivially satisfied.

6.2 Finding Subgraphs in $\mathbb{I}_{super}$

The problem of supergraph query processing has also received some attention (e.g., in [5, 44, 46, 6, 51]). In principle, any of these algorithms can be utilized for the task at hand within *i*GQ. However, we choose to propose a new approach, which is efficient yet simple and avoids the complexities and overheads involved in the above general approaches. The point is that we want a method for supergraph query processing that can easily fit within the framework of *i*GQ and perform both subgraph and supergraph query indexing and processing. Algorithm 1 shows how $\mathbb{I}_{super}$ is created. Briefly, $\mathbb{I}_{super}$ is a trie, storing features of queries. For each feature f it stores a pair $\{g_i, o\}$ for each graph g_i in which f appears, where o is its number of occurrences in g_i. For each g_i it also stores the number of distinct features ($NF[g_i]$) it contains.

Algorithm 2 illustrates how $\mathbb{I}_{super}$ identifies candidates CS that are potential subgraphs of query g. The idea is to find those graphs that contain *only* features included in the query graph g (lines 19–22; the check for $count(g_i)$ on line 20, ensures that all individual features of g_i are contained in g), and where for each such graph g_i a feature f occurs at most as many times as f occurs in g (line 12). Last, the graphs in CS are isomorphically tested to verify that $g_i \subseteq g$.

It is straightforward to see that no false negatives can exist in CS. Assume there is a false negative g_i such that $g_i \subseteq g$ and $g_i \notin CS$. Since $g_i \subseteq g$, any feature f in g_i appears no more times than f appears in g, thus g_i would be added to $\mathbb{G}$ on every execution of line 12. As $g_i \subseteq g$, all of g_i's features must appear in g. Thus, g_i would pass the if-clause at line 20 and be added to CS (contradiction). Moreover, subgraph isomorphism testing of all members of CS precludes false positives. Hence, formula (2)'s assumption holds.

Algorithm 2 Supergraph Query Processing in iGQ

```
1: Input: Query graph g and 𝕀_super
2: Output: Candidate set CS of potential subgraphs of g
3:
4: Initialize multiset 𝔾 = ∅
5: Extract all features of query graph g, F(g)
6: for all features f ∈ F(g) do
7:     O[f, g] = number of occurrences of f in g
8: end for
9: for all features f ∈ F(g) do
10:    if f ∈ 𝕀_super then
11:        for all {g_i, o} ∈ 𝕀_super.get(f) do
12:            if o ≤ O[f, g] then
13:                𝔾.insert(g_i)
14:            end if
15:        end for
16:    end if
17: end for
18: for all graphs g_i ∈ 𝔾 do
19:    count(g_i) = number of occurrences of g_i in 𝔾
20:    if count(g_i) == NF[g_i] then
21:        CS.add(g_i)
22:    end if
23: end for
24: return CS
```

6.3 iGQ System Operation

Fig. 6 depicts the complete *i*GQ system operation when used to expedite a subgraph query processing method $\mathbb{M}_{sub}$. Please keep in mind, though, that *i*GQ can be integrated with any subgraph and/or supergraph querying method. Given a new subgraph query g:

1. The query is sent to three separate processing threads in parallel and also stored in the query *window*.
2. In the first thread, $\mathbb{M}_{sub}$ uses its *Dataset Graph Index* to filter the dataset graphs and produce the candidate set $CS(g)$, as usual.
3. The remaining two threads perform filtering along the subgraph (section 4.2.1) and supergraph path (section 4.2.2). Their results are combined to prune $CS(g)$, based on formulae (3) and (5).
4. The resulting candidate set, $CS_{igq}(g)$, undergoes subgraph isomorphism testing to produce $Ans_{igq}(g)$.
5. Since this is for a subgraph query, the graphs pruned during processing along the subgraph path in step 3 are added to $Ans_{igq}(g)$ to produce the final answer set, $Answer(g)$ (see formula (4)).
6. Metadata maintained throughout the processing of g, including $Answer(g)$ and its subgraphs/supergraphs detected during step 3, are added to the metadata store, $Stat(iGQ\_Graph)$.
7. If the query window is full, the system uses the above metadata to select appropriate cached graphs to evict. Said graphs are replaced by the graphs in the window.
8. Finally, the *i*GQ index is updated to reflect the new contents of the cache (section 5.2 details the maintenance of the *i*GQ index).

7. PERFORMANCE EVALUATION

We have implemented the *i*GQ algorithms and report on experiments evaluating its performance on the savings of

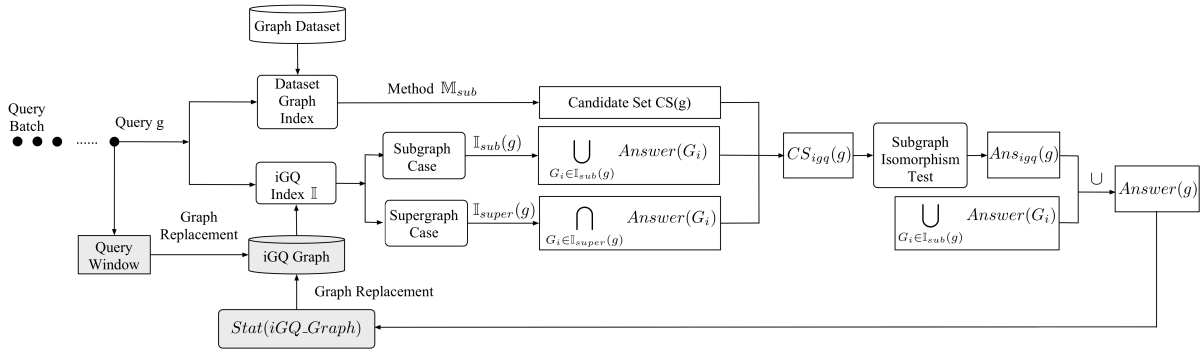

Figure 6: Operation of iGQ on Top of a Subgraph Query Processing Method $\mathbb{M}_{sub}$

subgraph query processing time (supergraph query processing time is omitted for space reason) and on the number of subgraph isomorphism tests.

7.1 Experimental Setup

Experiments were run on a Dell R920 system (4 Intel Xeon(R) CPUs (15 cores each), 512GB RAM, 1TB disk), and on a cluster of four Dell R720's (each with 2 Intel Xeon(R) CPUs (8 cores each), 64GB RAM, 1TB disk).

Algorithms. In addition to our implementation of iGQ query processing, we also secured access to implementations of three recent high performing subgraph query processing methods, GraphGrepSX[3] (GGSX), Grapes[15], and CT-Index[22]. In addition to their competitive performance, these three methods represent interesting design decisions. GGSX indexes paths (up to a certain maximum length, equal to 4 in these experiments) and uses the VF2 subgraph isomorphism algorithm for its verification stage. Grapes, like GGSX, also indexes paths (of up to length of 4), but utilizes location information in the filtering stage to expedite the verification stage, essentially focusing only on connected components of the dataset graphs that may contain the query graph. CT-Index indexes trees (of maximum size 6), and cycles (of maximum size 8) in hash-based bitmap structures (4096-bit wide), and uses a modified VF2 for its verification stage.

The implementations for Grapes and GGSX were obtained from the corresponding project web sites[15, 3]. For Grapes, we present two alternatives, Grapes and Grapes(6), which use 1 and 6 threads respectively. For fairness, we altered the code of Grapes so to stop query processing when the first match was found, instead of looking for all matches of a query within each stored graph. For CT-Index we obtained the JAR file from one of the authors, which we then reverse-engineered to derive its code in Java. Subsequently, we integrated the iGQ algorithms of Section 4.2.1 within Grapes, CT-Index, and GGSX, yielding three different versions of iGQ, denoted as iGQ_Grapes, iGQ_CT-Index, and iGQ_GGSX. In this way, (i) we validate our claim that iGQ can be incorporated into existing approaches, and (ii) we show that it can introduce significant performance gains during subgraph query processing of any of these approaches.

Datasets. We have employed three real-world datasets and one synthetic dataset with different characteristics, outlined in Table 1. AIDS is the Antiviral Screen Dataset of the National Cancer Institute, containing topological structures of molecules [30]. PDBS[19] is a dataset of graphs representing DNA, RNA and proteins. As AIDS and PDBS contain typical but relatively sparse graphs, we have performed further experiments on dense datasets, including the PPI dataset and a synthetic dataset. PPI[15] models large and dense protein interaction networks and consists of 20 graphs. We also used the generator provided by [7] to create a much larger number (1,000) of much denser graphs.

dataset	unique vertex labels	graphs in dataset	average node degree	num. nodes per graph			num. edges per graph		
				avg	std.dev	max	avg	std.dev	max
AIDS	62	40,000	2.09	45	22	245	47	23	250
PDBS	10	600	2.13	2,939	3,217	16,431	3,064	3,264	16,781
PPI	46	20	9.23	4,943	2,717	10,186	26,667	26,361	89,674
Synthetic	20	1,000	19.52	892	417	7,135	7,991	5	8,007

Table 1: Characteristics of Datasets

Query Workloads. Unfortunately, despite the availability of graph datasets, the community does not enjoy well established benchmarks and/or real-world query logs for these datasets. So all related works synthesize queries derived from components of the dataset graphs. We follow this established principle for generating our workloads, whereby queries are generated from the original dataset graphs as follows. There are 3 key probability distributions to consider here. The first governs how a graph is selected from the dataset graphs. The second governs how a node is selected within this graph. Given these, we produce 4 query workloads: $uni{-}uni$, $uni{-}zipf$, $zipf{-}uni$, and $zipf{-}zipf$, with, e.g., $zipf - uni$ denoting that dataset graphs have a popularity (probability of being selected) following a Zipf distribution, while nodes within the selected graph have a popularity drawn from a uniform distribution. The probability density function of the Zipf distribution is given by: $p(x) = \frac{x^{-\alpha}}{\zeta(\alpha)}$, where ζ is the Riemann Zeta function[31]. The default value for α was 1.4 – we have also used $\alpha = 1.1$ representing a much smaller skew and $\alpha = 2.0$ representing a stronger skew (as a reference point, web page popularities follow a Zipf with $\alpha = 2.4$ [31]). The third governs the size of each graph query: query sizes are uniformly at random selected from 4, 8, 12, 16, 20 edges. Once a graph and a node within this graph have been selected, we then perform a BFS traversal of the latter's neighborhood, with unvisited edges of each traversed node included in the generated graph, until the desired query size is reached.

For AIDS and PDBS, we ran 3,000 queries for each experiment. The first W of these queries were used to warm-up the index. We then used the remaining queries to measure the times and candidate set sizes with and without iGQ for each algorithm. By default we use a cache size $\mathcal{C} = 500$ and a batch window (and warm-up set) size $W = 100$ queries – we have also used $\mathcal{C} = 1000, W = 200$ and $\mathcal{C} = 1500, W = 300$ with a 5,000-query workload to test cache size impact. We

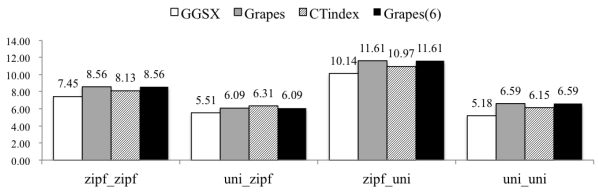

Figure 7: Speedup in Number of Subgraph Isomorphism Tests for AIDS

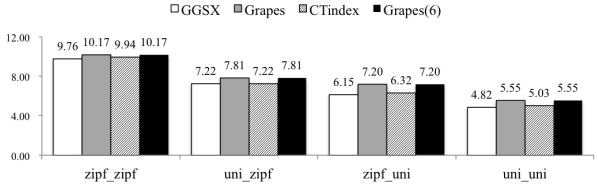

Figure 8: Speedup in Number of Subgraph Isomorphism Tests for PDBS

further tested iGQ against PPI and the synthetic dataset, in order to examine its performance under larger and denser graphs. In these cases, queries take 1-2 orders of magnitude more time to execute, hence for practical reasons we reduced the query workload to 500 queries. The batch window (and warm-up set) size were set to $W = 20$ queries, with cache sizes of $\mathcal{C} = 100, 200, 300$ and Zipf skew $\alpha = 1.4, 2.0, 2.4$.

We report the speedup (reduction) achieved by iGQ, defined as the ratio of the average performance of the traditional method $\mathbb{M}$ over the average performance of iGQ_$\mathbb{M}$, for the number of subgraph isomorphism tests and the query processing time.

7.2 Filtering Power Speedup

We first examine the *filtering power*, reflecting the speedup in the number of subgraph isomorphism tests performed. This metric facilitates a qualitative analysis of performance, independent of implementation and system details. Fig. 7 and 8 depict results for the AIDS and PDBS datasets respectively, across all four query workloads. The reduction in the number of subgraph isomorphism tests is evident (speedups of 5× to 11×). Fig. 9 shows how Zipf skew α affects this metric for the PDBS dataset, using one of the fastest methods (Grapes(6)). Results for the AIDS dataset and the other algorithms are similar and omitted for space reasons. As expected, with more skewness come increased benefits by iGQ.

Fig. 10 focuses on speedup across queries grouped by size (e.g., Q4 groups queries with 4 edges). As iGQ does not maintain separate caches per query size, the various query groups compete for the same space. Thus, some of them may seem to exhibit a lower speedup for larger cache sizes (e.g., the speedup of Q16 drops slightly when going from $\mathcal{C}$ =200 to 300); however, the speedup for the whole workload exhibited a steady rise (2.18, 2.45 and 2.53 for $\mathcal{C}$= 100, 200 and 300 respectively; figure omitted due to space reasons). Last, Fig. 11 depicts the results for the synthetic dataset.

7.3 Query Processing Speedup

Fig. 12 and 13 show the query processing time speedup for the AIDS/PDBS datasets. Interestingly, juxtaposing Fig. 12 against Fig. 7 (and Fig. 13 against Fig. 8) we see that reductions in the number of subgraph isomorphism tests do not directly translate into equal gains in query processing

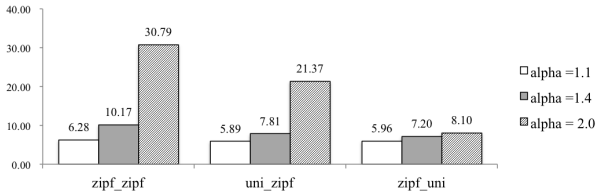

Figure 9: Speedup in Number of Subgraph Isomorphism Tests for PDBS/Grapes(6) vs Zipf Skew α

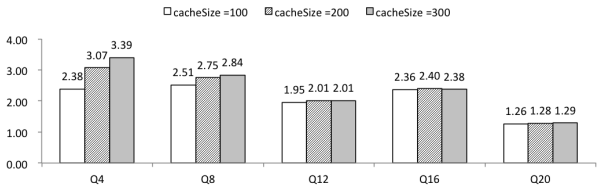

Figure 10: Speedup in Number of Subgraph Isomorphism Tests for PPI/Grapes(6)/$zipf - zipf(\alpha = 1.4)$/Query Groups

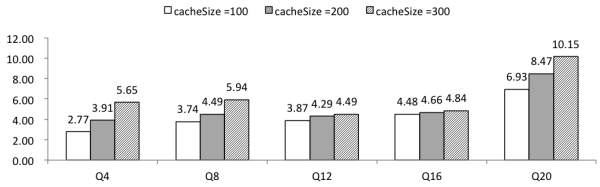

Figure 11: Speedup in Number of Subgraph Isomorphism Tests for Synthetic/Grapes(6)/$zipf - zipf(\alpha = 2.4)$/Query Groups

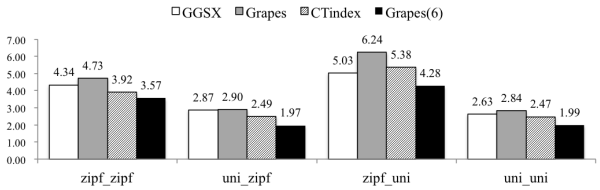

Figure 12: Speedup in Query Processing Time for AIDS

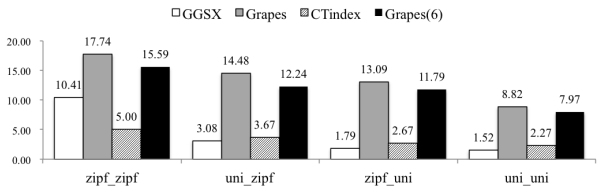

Figure 13: Speedup in Query Processing Time for PDBS

times. This is due to some large graphs in the candidate sets not being pruned away by the current index contents. We would expect this to be ameliorated as cache sizes increase. Indeed Fig. 14 shows this for Grapes(6) as cache size varies from 500 to 1,000 and 1,500 queries. Results for other cases are similar and omitted for space reasons.

Fig. 15 shows the impact of Zipf skew α on query processing speedup for the Grapes(6) algorithms on the PDBS dataset. Again, with more skewness come greater benefits, up to impressive levels. Interestingly, juxtaposing Fig. 15 against Fig. 9 tells a different story. We see that reductions in the number of subgraph isomorphism tests translate into higher gains in query processing times. This is because of the replacement algorithm that maintains in the index those

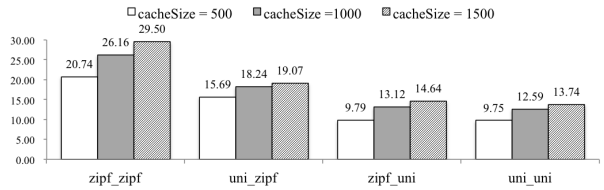

Figure 14: Speedup in Query Processing Time for PDBS/Grapes(6) vs Cache Size

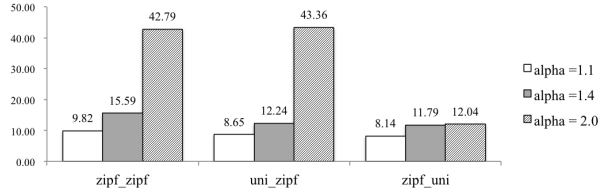

Figure 15: Speedup in Query Processing Time for PDBS/Grapes(6) vs Zipf Skew α

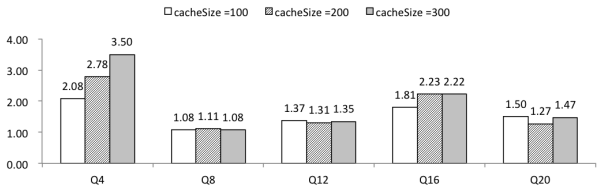

Figure 16: Speedup in Query Processing Time for PPI/Grapes(6)/$zipf - zipf(\alpha = 1.4)$/Query Groups

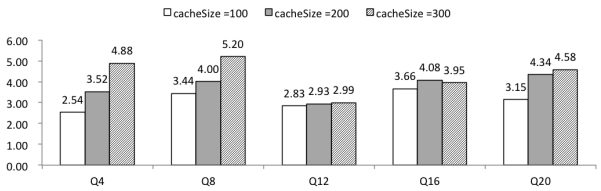

Figure 17: Speedup in Query Processing Time for Synthetic/Grapes(6)/$zipf - zipf(\alpha = 2.4)$/Query Groups

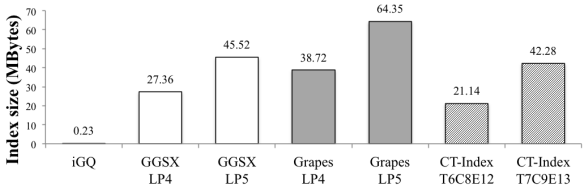

Figure 18: Absolute Index Sizes (in MBytes) for AIDS

query graphs g with the higher utility; i.e., which help to avoid expensive subgraph isomorphism tests against graphs in the candidate sets. Fig. 16 and Fig. 17 show the speedup for the query processing time, corresponding to Fig. 10 and Fig. 11, respectively.

Last, Fig. 18 plots the index size for iGQ for C =500 graph queries, versus that of the three algorithms we've considered so far, for the AIDS dataset. In the default configurations, iGQ adds a negligible space overhead on top of the base indexes (less than 1%). In addition to the default configurations for said algorithms, Fig. 18 also plots the index sizes for the immediately larger configurations (i.e., for max path length of 5 for Grapes and GGSX, and for trees of size 7, cycles of size 9, and 8192 bits per bitmap for CT-Index). Note that this minimal increase in the feature size results in almost double the space requirements for the base indexes. On the other hand, these larger indexes bring a performance improvement of less than 10% in all cases (figure omitted due to space reasons), which is virtually negligible when compared to the gains provided by iGQ.

Overall, iGQ is shown to introduce significant to impressive performance gains, against the state of the art methods in the literature. We have actually conducted a detailed performance evaluation of most related algorithms[21] and selected GGSX, Grapes(1), Grapes(6), and CT-Index as those showing the best performance. Regardless of the method, when incorporating iGQ with it, large performance gains ensue. These gains are robust and are manifested in all four different query workloads we have presented and, most importantly, with a minimal space overhead.

8. CONCLUSIONS

We have presented a novel perspective and solution to the graph querying problem, departing from related work in three ways: First, it constructs query indexes, as opposed to simply relying on dataset graph indexes. Second, it maintains the knowledge the system produced when executing previous queries. Third, it can be used to expedite both subgraph and supergraph queries. We showed how these can help improve the performance of future queries and provided formal proof of correctness. The proposed iGQ framework consists of (i) a subgraph index, (ii) a supergraph index, (iii) a method for efficiently maintaining the index, including a graph replacement policy, and (iv) any popular method for indexing and processing subgraph or supergraph queries. We incorporated iGQ within 3 popular methods from related work, showcasing its wide applicability. Last, our performance evaluation on both real-world and synthetic datasets with various query workloads showed iGQ's significant performance gains and negligible space overhead.

9. REFERENCES

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using Map-Reduce. In *Proc. IEEE ICDE*, pages 62–73, 2013.

[2] A. Balmin, F. Ozcan, S. K. Beyer, J. R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. VLDB*, pages 60–71, 2004.

[3] V. Bonnici, et al. Enhancing graph database indexing by suffix tree structure. In *Proc. IAPR PRIB*, 2010. `http://cs.nyu.edu/shasha/papers/graphgrep/`.

[4] R. V. Bruggen. *Learning Neo4j*. O'Reilly Media, 2013.

[5] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proc. VLDB*, 2007.

[6] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *VLDBJ*, 20(4):521–539, 2010.

[7] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *Proc. ACM SIGMOD*, 2007.

[8] L. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *Proc. ICIAP*, 1999.

[9] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento.

A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.

[10] W. de Nooy, A. Mrvar, and V. Batagelj. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press, 2005.

[11] K. Degtyarenko, J. Hastings, P. de Matos, and M. Ennis. Chebi: An open bioinformatics and cheminformatics resource. *Curr. Protoc. Bioinformatics*, 14(26):1–20, 2009.

[12] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha. Sing: Subgraph search in non-homogeneous graphs. *BMC bioinformatics*, 11(1):96, 2010.

[13] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. GRAMI: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7), 2014.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[15] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS One*, 8(10):e76911, 2013. `http://ferrolab.dmi.unict.it/GRAPES/`.

[16] R. Giugno and D. Shasha. GraphGrep: A fast and universal method for querying graphs. In *Proc. IEEE ICPR*, 2002.

[17] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.

[18] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Proc. IEEE ICDE*, 2006.

[19] Y. He, et al. Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex. *PNAS*, 99:10325–10329, 2002.

[20] InfiniteGraph. `http://www.objectivity.com/infinitegraph`.

[21] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *PVLDB*, 8(12), 2015.

[22] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. IEEE ICDE*, 2011.

[23] G. Koloniari and E. Pitoura. Partial view selection for evolving social graphs. In *Proc. GRADES*, 2013.

[24] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in MapReduce. *PVLDB*, 2015.

[25] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[26] K. Lillis and E. Pitoura. Cooperative XPath caching. *SIGMOD'08*, pages 327–338, 2008.

[27] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proc. ESEC/FSE*, 2005.

[28] G.Malewicz, et al. Pregel: a system for large-scale graph processing. In *Proc. ACM PODC*, 2009.

[29] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *Proc. VLDB*, pages 469–480, 2005.

[30] NCI - DTP AIDS antiviral screen dataset. `http://dtp.nci.nih.gov/docs/aids/aids_data.html`.

[31] M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, 2005.

[32] E. Petras and C. Faloutsos. Similarity searching in medical image databases. *IEEE TKDE*, 9(3), 1997.

[33] T. Plantenga. Inexact subgraph isomorphism in MapReduce. *J. Parallel Distrib. Comput.*, 73:164–175, 2013.

[34] PubChem. `https://pubchem.ncbi.nlm.nih.gov/`.

[35] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *Proc. IEEE ICDE*, 2016 (to appear).

[36] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[37] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proc. IEEE ICDE*, 2008.

[38] Twitter FlockDB. `https://github.com/twitter/flockdb`.

[39] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[40] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proc. IEEE ICDE*, 2007.

[41] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. ACM SIGMOD*, 2004.

[42] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based similarity search in graph structures. *ACM TODS*, 31(4):1418–1453, 2006.

[43] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *VLDBJ*, 22(2):229–252, 2013.

[44] D. Yuan, P. Mitra, and C. L. Giles. Mining and indexing graphs for supergraph search. *PVLDB*, 6(10), 2013.

[45] S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. In *Proc. IEEE ICDE*, 2007.

[46] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *Proc. EDBT*, 2009.

[47] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1-2):1185–1194, 2010.

[48] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 2010.

[49] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta >= graph. In *Proc. VLDB*, 2007.

[50] X. Zhao, et al. Efficient processing of graph similarity queries with edit distance constraints. *VLDBJ*, 22(6):727–752, Dec 2013.

[51] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang. Prefindex : An efficient supergraph containment search technique. In *Proc. SSDBM*, 2010.

[52] Y. Zhu, J. Yu, and L. Qin. Leveraging graph dimensions in online graph search. *PVLDB*, 8(1), 2014.

[53] L. Zou, L. Chen, J. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proc. ACM EDBT*, 2008.